25장. 클래스

25.1 클래스는 프로토타입의 문법적 설탕인가?

1. 자바스크립트는 프로토타입 기반(prototype based) 객체지향 언어다.

2. 클래스 기반 언어에 익숙한 프로그래머들은 프로토타입 기반 프로그래밍 방식에 혼란을 느낄 수 있으며, 자바스크립트를 어렵게 느끼게 하는 하나의 장벽처럼 인식되었다.

3. ES6에서 도입된 클래스는 클래스 기반 객체지향 프로그래밍에 익숙한 프로그래머가 더욱 빠르게 학습할 수 있도록, 클래스 기반 객체지향 프로그래밍 언어와 매우 흡사한 **새로운 객체 생성 메커니즘을 제시**한다.

4. 클래스는 함수이며 기존 프로토타입 기반 패턴을 클래스 기반 패턴처럼 사용할 수 있도록 하는 **문법적 설탕(syntactic sugar)**이라고 볼 수도 있다.

5. 클래스와 생성자 함수는 매우 유사하게 동작하지만 몇 가지 차이가 있다.

   > 1. 클래스를 `new` 연산자 없이 호출하면 에러가 발생한다.
   >
   >   (생성자 함수를 `new` 연산자 없이 호출하면 일반 함수로 호출된다.)
   > 2. 클래스는 상속을 지원하는 `extends`와 `super`키워드를 제공한다.
   >
   >   (생성자 함수는 지원하지 않는다.)
   > 3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작한다.
   >
   >   (함수 선언문은 함수 호이스팅, 함수 표현식은 변수 호이스팅이 발생한다.)
   > 4. 클래스 내의 모든 코드에는 암묵적으로 `strict mode`가 지정되어 실행되며 해제할 수 없다. (생성자 함수는 지정되지 않는다.)
   > 5. 클래스의 `constructor`, 프로토타입 메서드, 정적 메서드는 열거되지 않는다.
   >
   >   (모두 프로퍼티 어트리뷰트 `[[Enumerable]]`의 값이 `false`다.)

6. 생성자 함수와 클래스는 **프로토타입 기반의 객체지향을 구현했다는 점**에서 매우 유사하지만 클래스가 좀 더 견고하고 명료하다.

➡️ 따라서 클래스를 프로토타입 기반 객체 생성 패턴의 단순한 문법적 설탕이라고 보기보단,

새로운 객체 생성 메커니즘으로 보는 것이 좀 더 합당하다.

25.2 클래스 정의

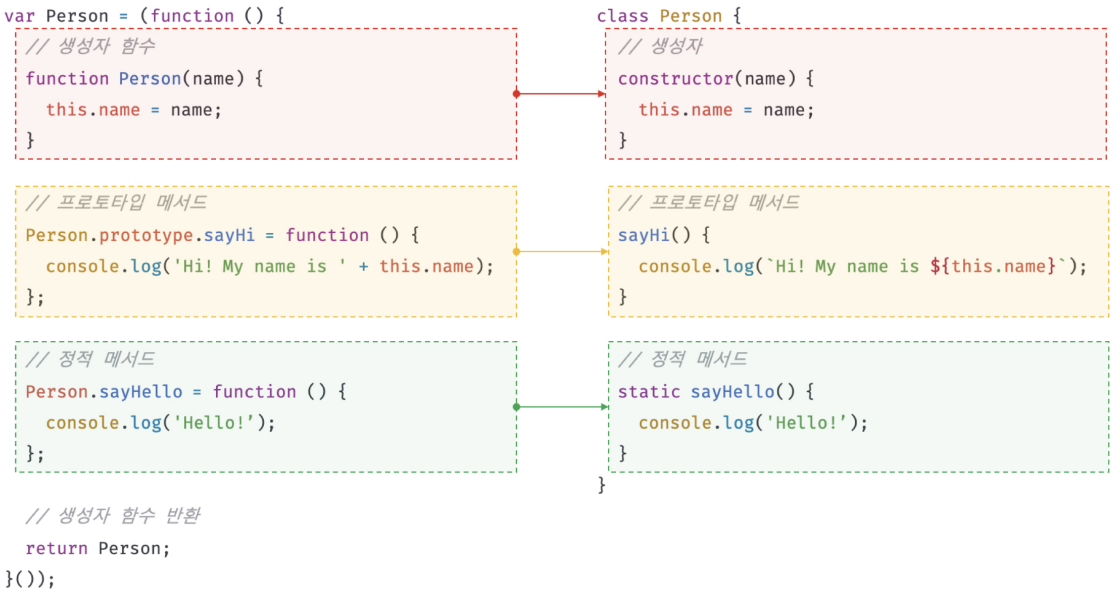
1. 클래스는 class 키워드를 사용하여 정의

   클래스 이름은 생성자 함수와 마찬가지로 파스칼 케이스를 사용하는 것이 일반적

2. [예제 25-04]

```
// 클래스 선언문
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name; // name 프로퍼티는 public
  }

  // 프로토타입 메서드
  sayHi() {
    console.log(`Hi! My name is ${this.name}`);
  }

  // 정적 메서드
  static sayHello() {
    console.log('Hello!');
  }
}

// 인스턴스 생성
const me = new Person('Lee');

// 인스턴스의 프로퍼티 참조
console.log(me.name); // Lee
// 프로토타입 메서드 호출
me.sayHi(); // Hi! My name is Lee
// 정적 메서드 호출
Person.sayHello(); // Hello!
```

그림 25-1



출처 : https://velog.io/@kozel/모던-자바스크립트-25장-클래스

- 클래스와 생성자 함수의 정의 방식은 형태적인 면에서 매우 유사

25.3 클래스 호이스팅

1. 클래스 선언문으로 정의한 클래스는 함수 선언문과 같이 소스코드 평가 과정,
   즉 런타임 이전에 먼저 평가되어 함수 객체를 생성한다.

2. 클래스는 클래스 정의 이전에 참조할 수 없다.

3. 클래스 선언문은 마치 호이스팅이 발생하지 않는 것처럼 보이나 그렇지 않다.

   [예제 25-07]

```
const Person = '';

{
  console.log(Person);
  // ReferenceError: Cannot access 'Person' before initialization

  class Person {}
}
```

- 클래스 선언문도 변수, 함수와 마찬가지로 호이스팅이 발생한다.
- 클래스는 let, const 키워드로 선언한 변수처럼 호이스팅된다.
  따라서 일시적 사각지대(TDZ)에 빠지기 때문에 호이스팅이 발생하지 않은 것처럼 동작한다.

25.4 인스턴스 생성

1. 클래스는 생성자 함수이며 new 연산자와 함께 호출되어 인스턴스를 생성

2. 클래스는 인스턴스를 생성하는 것이 유일한 존재 이유이므로
   반드시 **new 연산자**와 함께 호출해야 한다.

   [예제 25-09]

```
class Person {}

const me = Person();
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

3. 클래스 표현식으로 정의된 클래스의 경우, 클래스를 가리키는 식별자(Person)를 사용해 인스턴스를 생성하지 않고 기명 클래스 표현식의 클래스 이름(MyClass)을 사용해 인스턴스를 생성하면 에러 발생.

   [예제 25-10]

```
const Person = class MyClass {};

// 함수 표현식과 마찬가지로 클래스를 가리키는 식별자로 인스턴스를 생성해야 한다.
const me = new Person();

// 클래스 이름 MyClass는 함수와 동일하게 클래스 몸체 내부에서만 유효한 식별자다.
console.log(MyClass); // ReferenceError: MyClass is not defined

const you = new MyClass(); // // ReferenceError: MyClass is not define
d
```

25.5 메서드

25.5.1 constructor

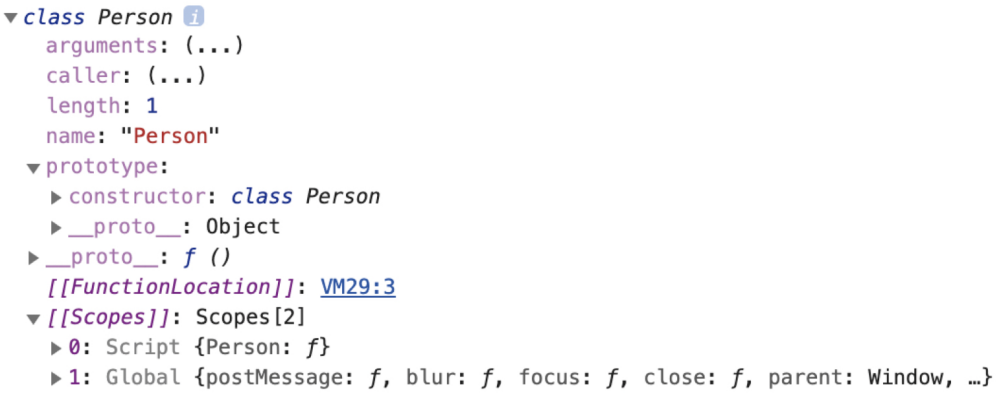
1. [예제 25-11]

```
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }
}
```

- constructor는 인스턴스를 생성 & 초기화하기 위한 특수한 메서드
- constructor는 이름을 변경할 수 없다.

2. [예제 25-12]

```
// 클래스는 함수다.
console.log(typeof Person); // function
console.dir(Person);
```



- 모든 함수 객체가 가지고 있는 prototype 프로퍼티가 가리키는

  프로토타입 객체의 constructor 프로퍼티는 클래스 자신을 가리킴
- 이는 클래스가 인스턴스를 생성하는 생성자 함수라는 것을 의미

3. [예제 25-13] - new 연산자와 함께 클래스 호출

```
// 인스턴스 생성
const me = new Person('Lee');
console.log(me);
```

- Person 클래스의 constructor 내부에서 this에 추가한 name 프로퍼티가 클래스가 생성한 인스턴스의 프로퍼티로 추가된 것을 확인

  즉, 생성자 함수와 마찬가지로 constructor 내부에서 this에 추가한 프로퍼티는 인스턴스 프로퍼티가 된다.

4. [예제 25-14]

```
// 클래스
class Person {
    constructor(name) {
        // 인스턴스 생성 및 초기화
        this.name = name;
    }
}

// 생성자 함수
function Person(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
}
```

- constructor 내부의 this는 생성자 함수와 마찬가지로 클래스가 생성한 인스턴스를 가리킴.

5. 그림 25-2, 그림 25-3 어디에도 constructor 메서드는 보이지 않는다.

이는 **클래스 몸체에 정의한 constructor가 단순한 메서드가 아니라는 것** 의미

> 💡 **클래스의 constructor 메서드와 프로토타입의 constructor 프로퍼티**
>
> : 둘은 이름이 같아 혼동하기 쉽지만 직접적인 관련 X, 프로토타입의 constructor 프로퍼티는 모든 프로토타입이 가지고 있는 프로퍼티이며, **생성자 함수**를 가리킴.

6. constructor는 생성자와 유사하지만 **몇 가지 차이** 존재

   a. constructor는 클래스 내 **최대 한 개**만 존재 가능, 2개 이상이면 문법 에러 발생

   b. constructor는 생략할 수 있다.

   → 생략하게 되면 암묵적으로 빈 constructor가 정의됨.

   → 빈 constructor에 의해 빈 객체 생성

   c. [예제 25-18]

```
class Person {
     constructor() {
         // 고정값으로 인스턴스 초기화
         this.name = 'Lee';
         this.address = 'Seoul'
     }
 }

 const me = new Person();
 console.log(me);    // Person {name: "Lee", address: "Seoul"}
```

   - 프로퍼티 추가되어 초기화된 인스턴스 생성하려면, constructor 내부에서 this에 인스턴스 프로퍼티를 추가

   d. [예제 25-19]

```
class Person {
     constructor(name, address) {
         // 인수로 인스턴스 초기화
         this.name = name;
         this.address = address;
     }
```

```
}

const me = new Person('Lee', 'Seoul');
console.log(me);    // Person {name: "Lee", address: "Seoul"}
```

- 인스턴스를 생성할 때 클래스 외부에서 인스턴스 프로퍼티의 초기값을 전달하려면,
  constructor에 매개변수를 선언하고 인스턴스를 생성할 때 초기값을 전달한다.
  이때 초기값은 constructor의 매개변수에게 전달된다.

7. 따라서 인스턴스를 초기화 하려면 constructor을 생략해서는 X

   또한 new 연산자와 함께 클래스가 호출되면 생성자 함수와 동일하게 암묵적으로 this,
   즉 인스턴스를 반환하기 때문에 constructor는 별도의 반환문을 갖지 않아야 한다.

8. 만약 this가 아닌 다른 객체를 명시적으로 반환하면
   인스턴스가 반환되지 못하고 return 문에 명시한 객체가 반환된다.

[예제 25-20]

```
class Person {
    constructor(name) {
        this.name = name;

        return {};
    }
}

// constructor에서 명시적으로 반환한 빈 객체가 반환된다.
const me = new Person('Lee');
console.log(me);    // {}
```

[예제 25-21]

```
class Person {
    constructor(name) {
        this.name = name;

        return 100;
    }
```

```
}

// 명시적으로 반환한 원시값은 무시되고 암묵적으로 this가 반환된다.
const me = new Person('Lee');
console.log(me);    // Person { name: "Lee" }
```

- 명시적으로 원시값을 반환하면 원시값 반환은 무시되고 암묵적으로 this가 반환된다.

  이는 클래스의 기본 동작을 훼손하는 것.

  ➡️ **따라서 constructor 내부에서 return 문을 반드시 생략해야 한다.**

25.5.2 프로토타입 메서드

1. 클래스 몸체에서 정의한 메서드는 생성자 함수에 의한 객체 생성 방식과는 다르게,

   클래스의 prototype 프로퍼티에 메서드를 추가하지 않아도 프로토타입 메서드가 된다.

[예제 25-23]

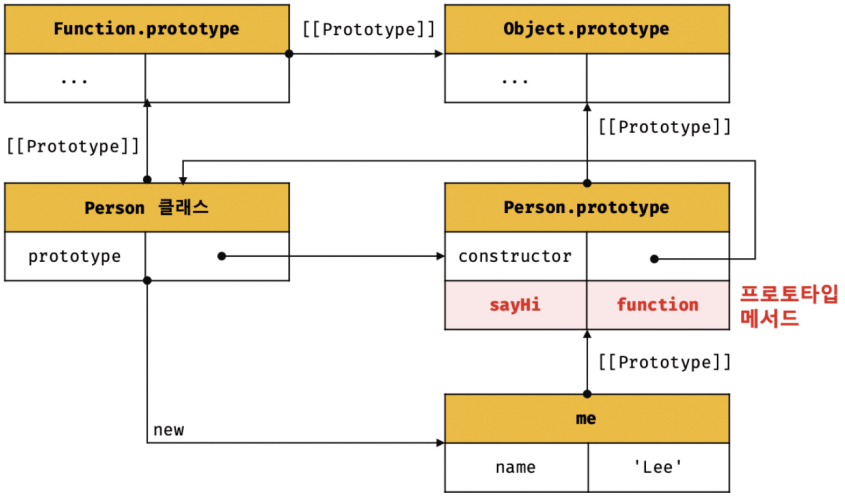
```
class Person {
    constructor(name) {
        this.name = name;
    }

    // 프로토타입 메서드
    sayHi() {
        console.log(`Hi My name is ${this.name}`);
    }
}
```

2. [예제 25-24]

```
// me 객체의 프로토타입은 Person.prototype이다.
Object.getPrototypeOf(me) === Person.prototype; // true
me instanceof Person;   // true

// Person.prototype의 프로토타입은 Object.prototype이다.
Object.getPrototypeOf(Person.prototype) === Object.prototype;   // tru
e
me instanceof Object;   // true

// me 객체의 constructor는 Person 클래스다.
me.constructor === Person;  // true
```

- 생성자 함수와 마찬가지로 클래스가 생성한 인스턴스는 프로토타입 체인의 일원이 된다.

3. 위 예제의 Person 클래스는 아래와 같이 프로토타입 체인을 생성



- 이처럼 인스턴스는 프로토타입 메서드를 상속받아 사용할 수 있다.
- 프로토타입 체인은 모든 객체 생성 방식(객체 리터럴, 생성자 함수, Object.create 메서드 등)뿐만 아니라 클래스에 의해 생성된 인스턴스에도 동일하게 적용된다.

➡️ 결국 클래스는 인스턴스를 생성하는 생성자 함수라고 볼 수 있고

클래스는 프로토타입 기반의 객체 생성 메커니즘이다.

25.5.3 정적 메서드

1. 정적 메서드는 인스턴스를 생성하지 않아도 호출할 수 있는 메서드를 말한다.

2. [예제 25-25]

```
function Person(name) {
    this.name = name;
}

// 정적 메서드
Person.sayHi = function () {
    console.log('Hi!');
};
```

```
// 정적 메서드 호출
Person.sayHi(); // Hi!
```

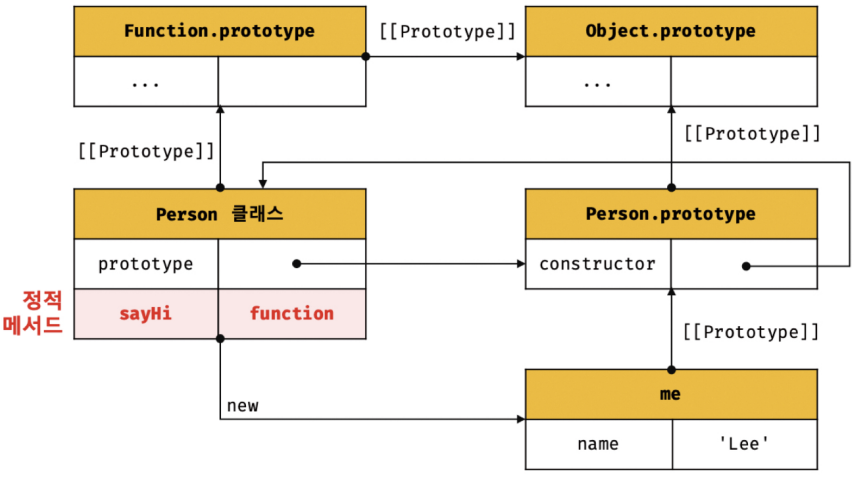
- 생성자 함수의 경우 정적 메서드를 생성하기 위해서는,

  명시적으로 생성자 함수에 메서드를 추가해야 한다.

3. [예제 25-26]

```
class Person {
    constructor(name) {
        this.name = name;
    }

    // 정적 메서드
    static sayHi() {
        console.log(`Hi!`);
    }
}
```

- 클래스에서는 메서드에 static 키워드를 붙이면 정적 메서드(클래스 메서드)가 된다.
- 위 예제의 Person 클래스는 다음과 같은 프로토타입 체인을 생성



- 이처럼 정적 메서드는 클래스에 바인딩된 메서드가 된다.
- 클래스는 함수 객체로 평가되므로 자신의 프로퍼티/메서드를 소유할 수 있다.

  클래스는 클래스 정의가 평가되는 시점에 함수 객체가 되므로 인스턴스와 달리 별다른 생성 과정이 필요 없다.
- 따라서 정적 메서드는 클래스 정의 이후 인스턴스를 생성하지 않아도 호출할 수 있다.

4. 정적 메서드는 클래스로 호출한다.

```
Person.sayHi(); // Hi
```

5. 인스턴스의 프로토타입 체인 상에는 클래스가 존재하지 않기 때문에,
   인스턴스로 클래스의 메서드를 상속받을 수 없다.

```
me.sayHi(); // TypeError: me.sayHi is not a function
```

25.5.4 정적 메서드와 프로토타입 메서드의 차이

1. 정적 메서드와 프로토타입 메서드의 차이
   a. 정적 메서드와 프로토타입 메서드는 자신이 속해 있는 프로토타입 체인이 다르다.
   b. 정적 메서드는 클래스로 호출하고, 프로토타입 메서드는 인스턴스로 호출한다.
   c. 정적 메서드는 인스턴스 프로퍼티를 참조할 수 없지만,
      프로토타입 메서드는 인스턴스 프로퍼티를 참조할 수 없다.

2. [예제 25-29]

```
class Square {
        // 정적 메서드
    static area(width, height) {
        return width * height;
    }
}

console.log(Square.area(10, 10));  // 100
```

- 정적 메서드 area는 인스턴스 프로퍼티를 참조하지 않는다.
  만약 참조해야 한다면 프로토타입 메서드를 사용해야 한다.

3. [예제 25-30]

```
class Square {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
```

```
  // 프로토타입 메서드
  area() {
    return this.width * this.height;
  }
}

const square = new Square(10, 10);
console.log(square.area()); // 100
```

- 정적 메서드는 클래스로 호출 → 정적 메서드 내부의 this는 인스턴스가 아닌 **클래스**를 가리킨다. 즉, 프로토타입 메서드와 정적 메서드 내부의 this 바인딩이 다르다.

  따라서 내부에서 인스턴스 프로퍼티를 참조할 필요가 있다면 프로토타입 메서드로 정의 후 this를 사용해야 한다.

- 물론 this를 사용하지 않더라도 프로토타입 메서드로 정의할 수 있다. 하지만 반드시 인스턴스를 생성한 다음 호출해야 하므로 this를 사용하지 않는다면 **정적 메서드**로 정의하는 것이 좋다.

4. 표준 빌트인 객체인 Math, Number, JSON, Object, Reflect 등이 가지는 정적 메서드는 애플리케이션 전역에서 사용할 유틸리티 함수다.

[예제 25-31]

```
Math.max(1, 2, 3);              // 3
Number.isNaN(NaN);              // true
JSON.stringify({ a: 1 });       //. "{"a":1}"
Object.is({}, {});              // false
Reflect.has({ a: 1 }, 'a');     // true
```

- 클래스 또는 생성자 함수를 하나의 네임스페이스로 사용하여 정적 메서드를 모아 놓으면,

  이름 충돌 가능성을 줄여 주고 관련 함수들을 구조화할 수 있는 효과가 있다.

▶ 정적 메서드는 애플리케이션 전역에서 사용할 유틸리티 함수를 전역 함수로 정의하지 않고 메서드로 구조화할 때 유용하다.

25.5.5 클래스에서 정의한 메서드의 특징

- 클래스에서 정의한 메서드는 다음과 같은 특징을 갖는다.
  a. function 키워드를 생략한 메서드 축약 표현을 사용한다.
  b. 객체 리터럴과는 다르게 클래스에 메서드를 정의할 때는 콤마가 필요 없다.
  c. 암묵적으로 strict mode로 실행된다.

d. `for...in` 문이나 Object.keys 메서드 등으로 열거할 수 없다. 즉, 프로퍼티 열거 가능 여부를 나타내며, 불리언 값을 갖는 프로퍼티 어트리뷰트 [[Enumerable]]의 값이 false다.

e. 내부 메서드 [[Construct]]를 갖지 않는 non-constructor다. 따라서 new 연산자와 함께 호출할 수 없다.

25.6 클래스의 인스턴스 생성 과정

- new 연산자와 함께 클래스를 호출하면 생성자 함수와 마찬가지로 클래스의 내부 메서드 [[Construct]]가 호출되면서 다음과 같은 과정을 거쳐 인스턴스가 생성

1. 인스턴스 생성과 this 바인딩

1. new 연산자와 함께 클래스를 호출하면 우선 암묵적으로 빈 객체, 바로 클래스가 생성한 인스턴스(아직 미완성)가 생성된다.
2. 클래스가 생성한 인스턴스의 프로토타입으로 클래스의 prototype 프로퍼티가 가리키는 객체가 설정된다.
3. 암묵적으로 생성된 빈 객체, 즉 인스턴스는 this에 바인딩된다.

   따라서 constructor 내부의 this는 클래스가 생성한 인스턴스를 가리킨다.

2. 인스턴스 초기화

1. constructor의 내부 코드가 실행되어 this에 바인딩되어 있는 인스턴스를 초기화한다.

- 즉, this에 바인딩되어 있는 인스턴스에 프로퍼티를 추가하고

  constructor가 인수로 전달 받은 초기 값으로 인스턴스의 프로퍼티 값을 초기화 한다.

- 만약 constructor가 생략되었다면 이 과정도 생략.

3. 인스턴스 반환

1. 클래스의 모든 처리가 끝나면 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.

- [예제 25-32]

```
class Person {
    constructor(name) {
        // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
        console.log(this);  // Parson {}
        console.log(Object.getPrototypeOf(this) === Person.prototype);
// true
```

```
        // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
        this.name = name;

        // 3. 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
    }
}
```

25.7 프로퍼티

25.7.1 인스턴스 프로퍼티

- 인스턴스 프로퍼티는 constructor 내부에서 정의해야 한다.

1. [예제 25-33]

```
class Person {
    constructor(name) {
        // 인스턴스 프로퍼티
        this.name = name;   // name 프로퍼티는 public하다.
    }
}
```

- constructor 내부 코드가 실행되기 이전에 constructor 내부의 this에는
  이미 클래스가 암묵적으로 생성한 인스턴스인 빈 객체가 바인딩되어 있다.
- constructor 내부에서 this에 추가한 프로퍼티는 언제나 클래스가 생성한 인스턴스의 프로퍼티가 된다.

25.7.2 접근자 프로퍼티

1. **접근자 프로퍼티**는 자체적으로는 값([[Value]] 내부 슬롯)을 갖지 않고
   다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 접근자 함수로 구성된 프로퍼티.

2. [예제 25-36]

```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
    // fullName은 접근자 함수로 구성된 접근자 프로퍼티다.
    // getter 함수
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    // setter 함수
    set fullName(name) {
        [this.firstName, this.lastName] = name.split(' ');
    }
}

const me = new Person('Ungmo', 'Lee');

// 접근자 프로퍼티를 통한 프로퍼티 값의 저장
me.fullName = 'Heegun Lee';
console.log(me);    // {firstName: "Heegun", lastName: "Lee"}

// 접근자 프로퍼티를 통한 프로퍼티 값의 참조
// 접근자 프로퍼티 fullName에 접근하면 getter 함수가 호출된다.
console.log(me.fullName);   // Heegun Lee

// fullName은 접근자 프로퍼티다.
// 접근자 프로퍼티는 get, set, enumerable, configurable 프로퍼티 어트리뷰트를
갖는다.
console.log(Object.getOwnPropertyDescriptor(Person.prototype, 'fullNam
e'));
// {get: f, set: f, enumerable: false, configurable: true}
```

- 접근자 프로퍼티는 자체적으로는 값을 갖지 않고 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 `getter` 함수와 `setter` 함수로 구성되어 있다.

getter

- 인스턴스 프로퍼티에 접근할 때마다 프로퍼티 값을 조작하거나 별도의 행위가 필요할 때 사용
- 메서드 이름 앞에 get 키워드를 사용해 정의
- 프로퍼티처럼 참조 시 내부적으로 getter가 호출
- 반드시 무언가를 반환해야 한다.

setter

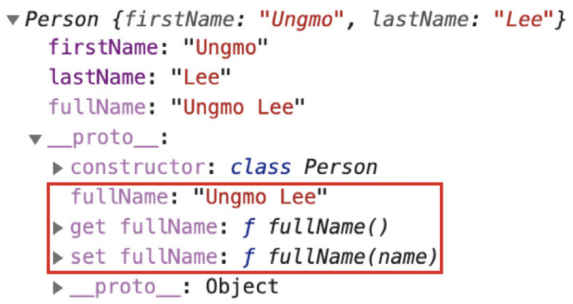
- 인스턴스 프로퍼티에 값을 할당할 때마다 프로퍼티 값을 조작 or 별도의 행위가 필요할 때 사용
- 메서드 이름 앞에 set 키워드를 사용해 정의
- 프로퍼티처럼 값을 할당하는 형식으로 하며 할당 시 내부적으로 setter가 호출

- 반드시 매개변수가 있어야 한다. 이때 단 하나의 매개변수만 선언할 수 있다.

3. getter와 setter 모두 인스턴스 프로퍼티처럼 사용된다.

4. 클래스의 메서드는 기본적으로 프로토타입 메서드가 되므로,
   접근자 프로퍼티 또한 인스턴스 프로퍼티가 아닌 프로토타입의 프로퍼티가 된다.

```
// Object.getOwnPropertyNames는 비열거형(non-enumerable)을 포함한 모든 프로퍼티 이름을
// 반환한다.(상속 제외)
Object.getOwnPropertyNames(me);                          // ["firstName", "lastName"]
Object.getOwnPropertyNames(Object.getPrototypeOf(me));   // ["constructor", "fullName"]
```



25.7.3 클래스 필드 정의 제안

- **클래스 필드**(필드 또는 멤버)
  : 클래스 기반 객체지향 언어에서 클래스가 생성할 인스턴스의 프로퍼티를 가리키는 용어

1. 클래스 기반 객체지향 언어(ex. Java)의 클래스
   - 클래스 기반 객체지향 언어의 this는 언제나 클래스가 생성할 인스턴스를 가리키고
     주로 클래스 필드가 생성자 또는 메서드의 매개변수 이름과 동일할 때 클래스 필드임을 명확히 하기 위해 사용된다.

2. 자바스크립트의 클래스 몸체에는 메서드만 선언할 수 있다. 따라서 클래스 몸체에 자바와 유사하게 클래스 필드를 선언하면 문법 에러(SyntaxError)가 발생한다.

3. [예제 25-39]

```
class Person {
  // 클래스 필드 정의
  name = 'Lee';
}

const me = new Person('Lee');
```

- 에러가 발생할 것 같지만 최신 브라우저(Chrome 72 이상) 또는 최신 Node.js(버전 12 이상)에서 실행하면 정상 동작
- 그 이유는 자바스크립트에서도 인스턴스 프로퍼티를 마치 클래스 기반 객체지향 언어의 클래스 필드처럼 정의할 수 있는 새로운 표준 사양인 "Class field declarations"가 TC39 프로세스의 stage3(candidate)에 제안되었기 때문

- Technical Committee 39(TC39)
: ECMA-262 사양의 관리를 담당하는 의원회이며 ECMA-262 사양(ECMAScript)을 제대로 준수해야 하는 기업으로 구성되어 있다.

- TC39 프로세스
: ECMA-262 사양에 새로운 표준 사양을 추가하기 위해 공식적으로 명문화해 놓은 과정을 말한다.

4. 클래스 몸체에서 클래스 필드를 정의하는 경우, this에 클래스 필드를 바인딩해서는 X

   this는 클래스의 constructor와 메서드 내에서만 유효

```
class Person {
    this.name = ''; // SyntaxError: Unexpected token '.'
}
```

5. 클래스 필드를 참조하는 경우 this를 반드시 사용해야 한다.

```
class Person {
  name = 'Lee';

  constructor() {
    console.log(name);  // ReferenceError: name is not defined
  }
}
```

```
new Person();
```

6. 클래스 필드에 초기값을 할당하지 않으면 undefined를 갖는다.

```
class Person {
  name;
}

const me = new Person();
console.log(me);    // Person {name: undefined}
```

7. 인스턴스를 생성할 때 외부의 초기값으로 클래스 필드를 초기화해야 할 필요가 있다면 constructor에서 클래스 필드를 초기화해야 한다.

```
class Person {
  name;

  constructor(name) {
    // 클래스 필드 초기화
    this.name = name;
  }
}

const me = new Person('Lee');
console.log(me);    // Person {name: "Lee"}
```

8. 인스턴스를 생성할 때 클래스 필드를 초기화할 필요가 있다면 constructor 밖에서 클래스 필드를 정의할 필요 X.

   클래스가 생성한 인스턴스에 클래스 필드에 해당하는 프로퍼티가 없다면 자동 추가되기 때문.

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const me = new Person('Lee');
console.log(me);    // Person {name: "Lee"}
```

9. 함수는 일급 객체이므로 함수를 클래스 필드에 할당할 수 있다.

   따라서 클래스 필드를 통해 메서드를 정의할 수도 있다.

```
class Person {
  name = 'Lee';

  // 클래스 필드에 함수를 할당
  getName = function () {
    return this.name;
  }
  // 화살표 함수로 정의할 수도 있다.
  // getName = () => this.name;
}

const me = new Person();
console.log(me);    // Person {name: "Lee", getName: f}
console.log(me.getName());  // Lee
```

10. 이처럼 클래스 필드에 함수를 할당하는 경우, 이 함수는 프로토타입 메서드가 아닌 인스턴스 메서드가 된다. 모든 클래스 필드는 인스턴스 프로퍼티가 되기 때문이다.

    따라서 **클래스 필드에 함수를 할당하는 것은 권장하지 않는다.**

25.7.4 private 필드 정의 제안

1. 자바스크립트는 캡슐화를 완전하게 지원하지 않는다. 클래스도 private, public, protected 키워드와 같은 접근 제한자를 지원하지 않는다.

   따라서 인스턴스 프로퍼티는 인스턴스를 통해 클래스 외부에서 언제나 참조할 수 있다. 즉, 언제나 public하다.

   다행히도 TC39 프로세스의 stage 3(candidate)에는 private 필드를 정의할 수 있는 새로운 표준 사양이 제안되었다.

2. 다음과 같이 private 필드의 선두에는 #을 붙여주며 참조할 때도 #을 붙여주어야 한다.

```
class Person {
  // private 필드 정의
  #name = '';

  constructor(name) {
    // private 필드 참조
```

```
    this.#name = name;
  }
}

const me = new Person('Lee');
console.log(me.#name);  // SyntaxError: Private field '#name' must be
declared in an enclosing class
```

3. public 필드는 참조할 수 있지만, private 필드는 클래스 내부에서만 참조할 수 있다.

접근 가능성	public	private
클래스 내부	O	O
자식 클래스 내부	O	X
클래스 인스턴스를 통한 접근	O	X

4. 클래스 외부에서 private 필드에 직접 접근할 수는 없지만

   접근자 프로퍼티를 통해 간접적으로 접근하는 방법은 유효하다.

```
class Person {
  #name = '';

  constructor(name) {
    this.#name = name;
  }

  get name() {
    return this.#name.trim();
  }
}

const me = new Person('Lee');
console.log(me.name);   // Lee
```

💡 **String.prototype.trim()**

`trim()` 메서드는 문자열 양 끝의 공백을 제거하고 원본 문자열을 수정하지 않고 새로운 문자열을 반환합니다. 여기서 말하는 공백이란 모든 공백문자(space, tab, NBSP 등)와 모든 개행문자(LF, CR 등)를 의미합니다.

5. private 필드는 반드시 클래스 몸체에 정의해야 한다. constructor에 정의하면 에러 발생

```
class Person {
  constructor(name) {
    // private 필드는 클래스 몸체에서 정의해야 한다.
    this.#name = name; // SyntaxError:
  }
}
```

25.7.5 static 필드 정의 제안

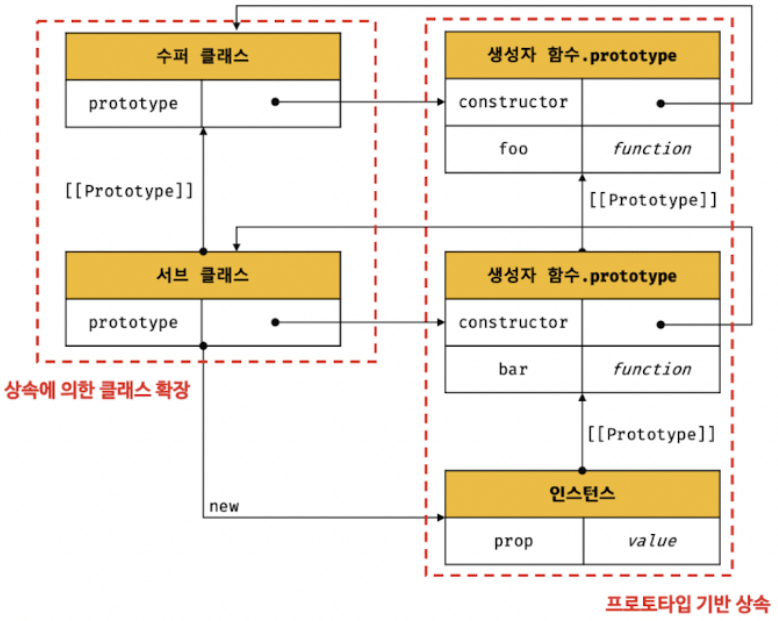
1. static 키워드를 사용하여 정적 필드를 정의할 수는 없었지만 static public 필드, static private 필드, static private 메서드를 정의할 수 있는 **새로운 표준 사양인 "Static class features"**가 TC39 프로세스의 stage 3(candidate)에 제안되었다.
    - 21년 기준 최신 브라우저에 구현되어 있음

```
class MyMath {
    // static public 필드 정의
    static PI = 22 / 7;

    // static private 필드 정의
    static #num = 10;

    // static 메서드
    static increment() {
        return ++MyMath.#num;
    }
}

console.log(MyMath.PI);           // 3.142857142857143
console.log(MyMath.increment());  // 11
```

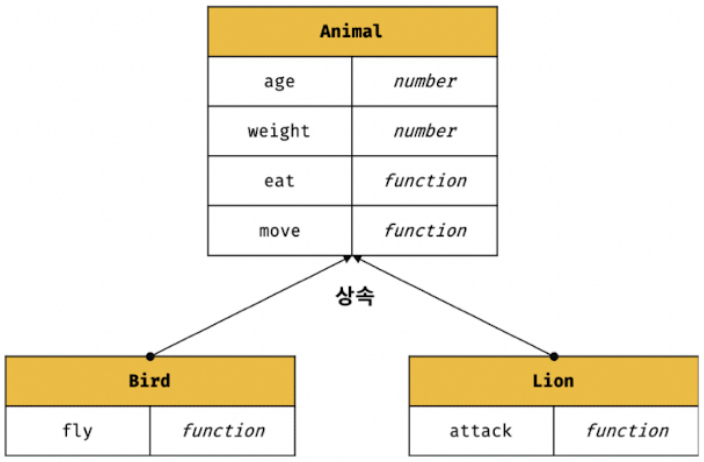
25.8 상속에 의한 클래스 확장

25.8.1 클래스 상속과 생성자 함수 상속

1. 프로토타입 기반 상속은 프로토타입 체인을 통해 다른 객체의 자산을 상속받는 개념이지만 **상속에 의한 클래스 확장은 기존 클래스를 상속받아 새로운 클래스를 확장하여 정의**하는 것이다.



2. 클래스는 상속을 통해 기존 클래스를 확장할 수 있는 문법이 기본적으로 제공되지만 생성자 함수는 그렇지 않다. 이런 상속을 통해 클래스의 속성을 그대로 사용하면서 자신만의 고유한 속성만 추가하여 확장할 수 있다.



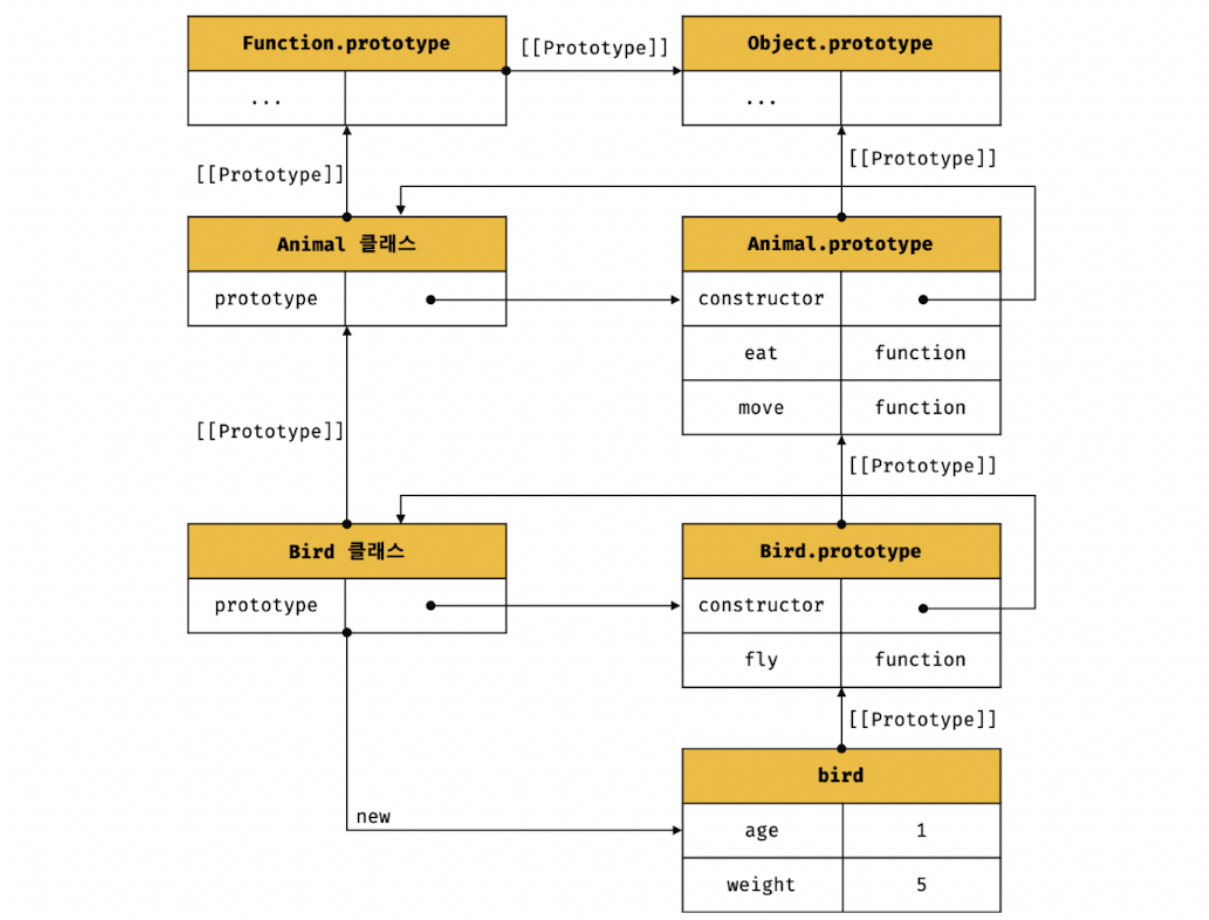
- 클래스 확장은 코드 재사용 관점에서 매우 유용하다.

3. [예제 25-54] - 상속을 통해 Animal 클래스를 확장한 Bird 클래스를 구현

```
class Animal {
    constructor(age, weight) {
        this.age = age;
        this.weight = weight;
```

```
    }

    eat() { return 'eat'; }

    move() { return 'move'; }
}

// 상속을 통해 Animal 클래스를 확장한 Bird 클래스
class Bird extends Animal {
    fly() { return 'fly'; }
}

const bird = new Bird(1, 5);

console.log(bird);                      // Bird {age: 1, weight: 5}
console.log(bird instanceof Bird);      // true
console.log(bird instanceof Animal);    // true

console.log(bird.eat());    // eat
console.log(bird.move());   // move
console.log(bird.fly());    // fly
```

- 상속에 의해 확장된 클래스 Bird를 통해 생성된 인스턴스의 프로토타입 체인은 다음과 같다.



- extends 키워드를 사용한 클래스 확장은 간편하고 직관적.

  하지만, 생성자 함수는 클래스와 같이 상속을 통해 다른 생성자 함수를 확장할 수 있는 문법이 제공되지 않는다.

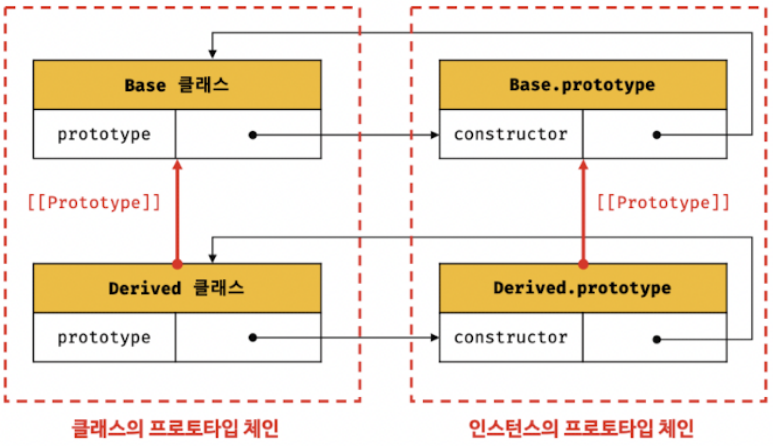
25.8.2 extends 키워드

1. 상속을 통해 클래스를 확장하려면 extends 키워드를 사용하여 상속받을 클래스를 정의.

```
// 수퍼(베이스/부모)클래스
class Base {}

// 서브(파생/자식)클래스
class Derived extends Base {}
```

2. extends 키워드의 역할은 수퍼클래스와 서브클래스 간의 상속 관계를 설정하는 것.

   클래스도 프로토타입을 통해 상속 관계를 구현한다.



- 수퍼클래스와 서브클래스는 인스턴스의 프로토타입 체인뿐 아니라 클래스 간의 프로토타입 체인도 생성한다. 이를 통해 프로토타입 메서드, 정적 메서드 모두 상속이 가능하다.

25.8.3 동적 상속

1. extends 키워드는 클래스뿐만 아니라 생성자 함수를 상속받아 클래스를 확장할 수 있다.

   단, extends 키워드 앞에는 반드시 클래스가 와야 한다.

```
// 생성자 함수
function Base(a) {
    this.a = a;
}

// 생성자 함수를 상속받는 서브클래스
class Derived extends Base {}

const derived = new Derived(1);
console.log(derived);   // Derived {a: 1}
```

2. extends 키워드 다음에는 클래스뿐만 아니라 [[Construct]] 내부 메서드를 갖는 함수 객체로 평가될 수 있는 모든 표현식을 사용할 수 있다. 이를 통해 동적으로 상속받을 대상을 결정할 수 있다.

```
function Base1() {}

class Base2 {}

let condition = true;

// 조건에 따라 동적으로 상속 대상을 결정하는 서브 클래스
class Derived extends (condition ? Base1 : Base2) {}
```

```
const derived = new Derived();
console.log(derived);  // Derived {}

console.log(derived instanceof Base1);  // true
console.log(derived instanceof Base2);  // false
```

25.8.4 서브클래스의 constructor

1. 클래스에서 constructor를 생략하면 클래스에 비어있는 constructor가 암묵적으로 정의된다.

2. 서브클래스에서 constructor를 생략하면 클래스에 다음과 같은 constructor가 암묵적으로 정의된다. args는 new 연산자와 함께 클래스를 호출할 때 전달한 인수의 리스트다.

   `constructor(...args) { super(...args); }`

   - super()는 수퍼클래스의 constructor(super-constructor)를 호출하여 인스턴스를 생성한다.

3. 수퍼클래스와 서브클래스 모두 constructor를 생략했다.

```
// 수퍼클래스
class Base {}

// 서브클래스
class Derived extends Base {}
```

4. 위 예제의 클래스에는 다음과 같이 암묵적으로 constructor가 정의된다.

```
// 수퍼클래스
class Base {
    constructor() {}
}

// 서브클래스
class Derived extends Base {
    constructor(...args) { super(...args); }
}

const derived = new Derived();
console.log(derived);   // Derived {}
```

- 위와 같이 수퍼클래스와 서브클래스 모두 constructor를 생략하면 빈 객체가 생성된다. 프로퍼티를 소유하는 인스턴스를 생성하려면 constructor 내부에서 인스턴스에 프로퍼티를 추가해야 함.

25.8.5 super 키워드

1. super 키워드는 함수처럼 호출할 수도 있고 this와 같이 식별자처럼 참조할 수 있는 특수한 키워드다. super는 다음과 같이 동작한다.
   - super를 호출하면 수퍼클래스의 constructor(super-constructor)를 호출한다.
   - super를 참조하면 수퍼클래스의 메서드를 호출할 수 있다.

super 호출

- super를 호출하면 수퍼클래스의 constructor(super-constructor)를 호출한다.
- 다음 예제와 같이 수퍼클래스에서 추가한 프로퍼티와 서브클래스에서 추가한 프로퍼티를 갖는 인스턴스를 생성한다면 서브클래스의 constructor를 생략할 수 없다.
- 이때 new 연산자와 함께 서브클래스를 호출하면서 전달한 인수 중에서 수퍼클래스의 constructor에 전달할 필요가 있는 인수는 서브클래스의 constructor에서 호출하는 super를 통해 전달한다.

```
class Base {
    constructor(a, b) { // ④
        this.a = a;
        this.b = b;
    }
}

class Derived extends Base {
    constructor(a, b, c) {  // ②
        super(a, b);        // ③
        this.c = c;
    }
}

const derived = new Derived(1, 2, 3);   // ①
console.log(derived);                   // Derived {a: 1, b: 2, c: 3}
```

- 순서

  : new 연산자와 함께 Derived 클래스를 호출(①)하면서 전달한 인수 1, 2, 3은 Derived 클래스의 constructor(②)에 전달되고 super 호출(③)을 통해 Base 클래스의 constructor(④)에 일부가 전달된다.

2. super를 호출할 때 **주의할 사항**은 다음과 같다.

a. **서브클래스에서 constructor를 생략하지 않는 경우 서브클래스의 constructor에서는 반드시 super를 호출해야 한다.**

```
class Base {}

 class Derived extends Base {
    constructor() {
        // ReferenceError: Must call super constructor in derived c
lass before accessing 'this' or returning from derived constructor
        console.log('constructor call');
    }

 }

 const derived = new Derived();
```

b. **서브클래스의 constructor에서 super를 호출하기 전에는 this를 참조할 수 없다.**

```
class Base {}

 class Derived extends Base {
    constructor() {
        // ReferenceError: Must call super constructor in derived c
lass before accessing 'this' or returning from derived constructor
        this.a = 1;
        super();
    }
 }

 const derived = new Derived(1);
```

c. **super는 반드시 서브클래스의 constructor에서만 호출한다. 서브클래스가 아닌 클래스의 constructor나 함수에서 super를 호출하면 에러가 발생한다.**

```
class Base {
    constructor() {
        super();    // SyntaxError: 'super' keyword unexpected here
    }
 }

 function Foo() {
```

```
    super();    // SyntaxError: 'super' keyword unexpected here
}
```

super 참조

메서드 내에서 super를 참조하면 수퍼클래스의 메서드를 호출할 수 있다.

1. **서브클래스의 프로토타입 메서드 내에서 super.sayHi는 수퍼클래스의 프로토타입 메서드 sayHi를 가리킨다.**

```
class Base {
    constructor(name) {
        this.name = name;
    }

    sayHi() {
        return `Hi ${this.name}`;
    }
}

class Derived extends Base {
    sayHi() {
        // super.sayHi는 수퍼클래스의 프로토타입 메서드를 가리킨다.
        return `${super.sayHi()}. how are you doing?`;
    }
}

const derived = new Derived('Lee');
console.log(derived.sayHi());   // Hi Lee. how are you doing?
```

- super 참조를 통해 수퍼클래스의 메서드를 참조하려면 super가 수퍼클래스의 메서드가 바인딩된 객체, 즉 수퍼클래스의 prototype 프로퍼티에 바인딩된 프로토타입을 참조할 수 있어야 한다.

- 위 예제는 다음 예제와 동일하게 동작한다.

```
class Base {
    constructor(name) {
        this.name = name;
    }

    sayHi() {
        return `Hi ${this.name}`;
    }
}
```

```
class Derived extends Base {
   sayHi() {
       // __super는 Base.prototype을 가리킨다.
       const __super = Object.getPrototypeOf(Derived.prototype);
       return `${__super.sayHi.call(this)} how are you doing?`;
   }
}
```

- super는 자신을 참조하고 있는 메서드가 바인딩되어 있는 객체의 프로토타입을 가리킨다.

  따라서 super.sayHi, 즉 Base.prototype.sayHi를 호출할 때 call 메서드를 사용해 this를 전달해야 한다.

- call 메서드를 사용해 this를 전달하지 않고 Base.prototype.sayHi를 그대로 호출하면 Base.prototype.sayHi 메서드 내부의 this는 Base.prototype을 가리킨다.

  Base.prototype.sayHi 메서드는 프로토타입 메서드이기 때문에 내부의 this는 Base.prototype이 아닌 **인스턴스를 가리켜야 한다. name 프로퍼티는 인스턴스에 존재하기 때문이다.**

- 이처럼 super 참조가 동작하기 위해서는 super를 참조하고 있는 메서드가 바인딩되어 있는 객체의 프로토타입을 찾을 수 있어야하며 이를 위해 메서드는 내부 슬롯 [[HomeObject]]를 가지며, 자신을 바인딩하고 있는 객체를 가리킨다.

- super 참조를 의사 코드로 표현하면 다음과 같다.

```
/*
 [[HomeObject]]는 메서드 자신을 바인딩하고 있는 객체를 가리킨다.
 [[HomeObject]]를 통해 메서드 자신을 바인딩하고 있는 객체의 프로토타입을 찾을
수 있다.
 */
 super = Object.getPrototypeOf([[HomeObject]])
```

주의할 것은 ES6의 메서드 축약 표현으로 정의된 함수만이 [[HomeObject]]를 갖는다는 것이다.

```
 const obj = {
   // foo는 ES6의 메서드 축약 표현으로 정의한 메서드다. 따라서 [[HomeObjec
t]]를 갖는다.
   foo() {},
   // bar는 ES6의 메서드 축약 표현으로 정의한 메서드가 아니라 일반 함수다.
   // 따라서 [[HomeObject]]를 갖지 않는다.
   bar: function () {}
 };
```

- [[HomeObject]]를 가지는 ES6의 메서드 축약 표현으로 정의된 함수만이 super 참조를 할 수 있다. 단, super 참조는 수퍼클래스의 메서드를 참조하기 위해 사용하므로 서브클래스의 메서드에서 사용해야 한다.

```
const base = {
   name: 'Lee',
   sayHi() {
     return `Hi! ${this.name}`;
   }
 };

 const derived = {
   __proto__: base,
   // ES6 메서드 축약 표현으로 정의한 메서드다. 따라서 [[HomeObject]]를 갖는다.
   sayHi() {
     return `${super.sayHi()}. how are you doing?`;
   }
 };

 console.log(derived.sayHi());  // Hi Lee. how are you doing?
```

2. **서브클래스의 정적 메서드 내에서 super.sayHi는 수퍼클래스의 정적 메서드 sayHi를 가리킨다.**

```
class Base {  // 수퍼 클래스
   static sayHi() {
     return 'Hi!';
   }
 }

 class Derived extends Base {  // 서브 클래스
   static sayHi() {
     // super.sayHi는 수퍼클래스의 정적 메서드를 가리킨다.
     return `${super.sayHi()} how are you doing?`;
   }
 }

 console.log(Derived.sayHi());  // Hi how are you doing?
```

25.8.6 상속 클래스의 인스턴스 생성 과정

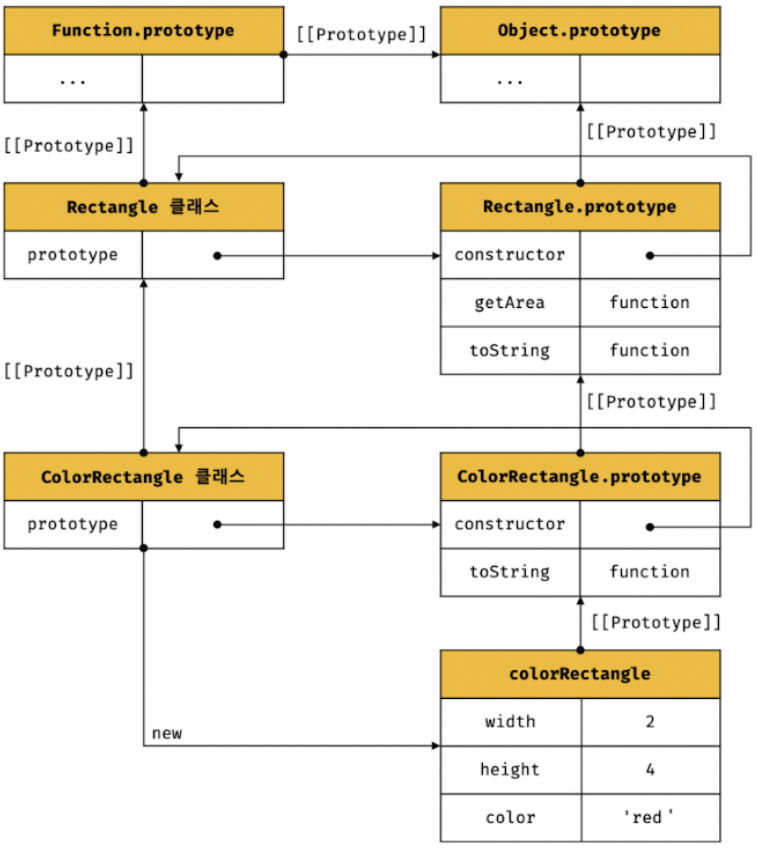
- 클래스가 단독으로 인스턴스를 생성하는 과정보다,

  상속 관계에 있는 두 클래스가 협력하여 인스턴스를 생성하는 과정은 좀 더 복잡하다.

1. 다음의 Rectangle 클래스와 상속받은 ColorRectangle 클래스를 보자.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }

    toString() {
        return `width = ${this.width}, height = ${this.height}`;
    }
}

class ColorReactangle extends Rectangle {
    constructor(width, height, color) {
        super(width, height);
        this.color = color;
    }

    // 메서드 오버라이딩
    toString() {
        return super.toString() + `, color = ${this.color}`;
    }
}

const colorReactangle = new ColorReactangle(2, 4, 'red');
console.log(colorReactangle);   // ColorRectangle {width: 2, height:
4, color: "red"}

console.log(colorReactangle.getArea());     // 8
console.log(colorReactangle.toString());    // width = 2, height = 4,
color = red
```

- colorRectangle 클래스에 의해 생성된 인스턴스의 프로토타입 체인은 다음과 같다.



: 서브클래스 colorRectangle이 new 연산자와 함께 호출되면 다음 과정을 통해 인스턴스를 생성한다.

1. **서브클래스의 super 호출**
   - 자바스크립트 엔진은 클래스를 평가할 때 수퍼클래스와 서브클래스를 구분하기 위해 "base" 또는 "derived"를 값으로 갖는 내부 슬롯 [[ConstructorKind]]를 갖는다.

     다른 클래스를 상속받지 않는 클래스(그리고 생성자 함수)는 내부 슬롯 [[ConstructorKind]]의 값이 "base"로 설정되지만, 다른 클래스를 상속받는 서브클래스 "derived"로 설정된다. 그리고 이를 통해 호출되었을 때의 동작이 구분된다.
   - 상속받지 않는 클래스는 new 연산자와 함께 호출되었을 때 암묵적으로 빈 객체,

     즉 인스턴스를 생성하고 이를 this에 바인딩한다.
   - 하지만 **서브클래스는 자신이 직접 인스턴스를 생성하지 않고 수퍼클래스에서 인스턴스 생성을 위임한다. 이것이 바로 서브클래스의 constructor에서 반드시 super을 호출해야 하는 이유다.**
   - 서브클래스가 new 연산자와 함께 호출되면 서브클래스 constructor 내부의 super 키워드가 함수처럼 호출되고 super가 호출되면 수퍼클래스가 평가되어 생성된 함수 객체의 코드가 실행되기 시작한다.
   - 서브클래스 constructor 내부에 super 호출이 없으면 에러가 발생한다.

     실제로 인스턴스를 생성하는 주체는 수퍼클래스이므로 수퍼클래스의 constructor를 호출하는 super가 호출되지 않으면 인스턴스를 생성할 수 없기 때문이다.

2. **수퍼클래스의 인스턴스 생성과 this 바인딩**

- 수퍼클래스의 constructor 내부의 코드가 실행되기 이전에 암묵적으로 빈 객체를 생성한다. 이 빈 객체가 바로 클래스가 생성한 인스턴스다. 그리고 암묵적으로 생성된 빈 객체, 즉 인스턴스는 this에 바인딩된다.

```
class Rectangle {
    constructor(width, height) {
      // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된다.
      console.log(this);    // ColorRectangle {}
      // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRectangle이
다.
      console.log(new.target);  // ColorRectangle
...
```

- 이때 인스턴스는 수퍼클래스가 생성한 것이다. 하지만 new 연산자와 함께 호출된 클래스가 서브클래스라는 것이 중요하다. 즉, new 연산자와 함께 호출된 함수를 가리키는 new.target은 서브클래스를 가리킨다.

  따라서 **인스턴스는 new.target이 가리키는 서브클래스가 생성한 것으로 처리된다.**

- 따라서 생성된 인스턴스의 프로토타입은 수퍼클래스의 prototype 프로퍼티가 가리키는 객체가 아니라 new.target, 즉 서브클래스의 prototype 프로퍼티가 가리키는 객체이다.

```
class Rectangle {
    constructor(width, height) {
      // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된다.
      console.log(this);    // ColorRectangle {}

      // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRectangle이
다.
      console.log(new.target);  // ColorRectangle

            // 생성된 인스턴스의 프로토타입으로 ColorRectangle.prototype이
설정된다.
      console.log(Object.getPrototypeOf(this) === ColorRectangle.pr
ototype);// true
      console.log(this instanceof ColorRectangle);  // true
      console.log(this instanceof Rectangle);       // true
...
```

3. **수퍼클래스의 인스턴스 초기화**

- 수퍼클래스의 `constructor`가 실행되어 `this`에 바인딩되어 있는 인스턴스를 초기화한다.

```
class Rectangle {
    constructor(width, height) {
      // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된다.
      console.log(this);    // ColorRectangle {}
      // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRectangle이
다.
      console.log(new.target);  // ColorRectangle

      console.log(Object.getPrototypeOf(this) === ColorRectangle.pr
ototype);   // true
      console.log(this instanceof ColorRectangle);  // true
      console.log(this instanceof Rectangle);       // true

      // 인스턴스 초기화
      this.width = width;
      this.height = height;

      console.log(this);    // ColorRectangle {width: 2, height: 4}
...
```

4. **서브클래스 constructor로의 복귀와 this 바인딩**

   `super`의 호출이 종료되고 제어 흐름이 서브클래스 `construcotr`로 돌아온다.

   이때 `super`**가 반환한 인스턴스가** `this`**바인딩된다.**

   서브클래스는 별도의 인스턴스를 생성하지 않고 `super`가 반환한 인스턴스를 그대로 사용한다.

```
class ColorRectangle extends Rectangle {
  constructor(width, height, color) {
    super(width, height);

    // super가 반환한 인스턴스가 this에 바인딩된다.
    console.log(this);  // ColorRectangle {width: 2, height: 4}
...
```

   이처럼 super가 호출되지 않으면 인스턴스가 생성되지 않으며, this 바인딩도 할 수 없다. 서브클래스의 constructor에서 super를 호출하기 전에는 this를 참조할 수 없는 이유가 바로 이 때문이다.

5. **서브클래스의 인스턴스 초기화**

   `super`호출 이후 서브클래스의 `constructor`에 기술되어 있는 인스턴스 초기화가 실행된다.

6. **인스턴스 반환**

클래스의 모든 처리가 끝나면 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.

```
class ColorRectangle extends Rectangle {
  constructor(width, height, color) {
    super(width, height);

    // super가 반환한 인스턴스가 this에 바인딩된다.
    console.log(this);  // ColorRectangle {width: 2, height: 4}

    // 인스턴스 초기화
    this.color = color;

    // 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
    console.log(this);  // ColorRectangle {width: 2, height: 4, col
or: "red"}
  }
...
```

25.8.7 표준 빌트인 생성자 함수 확장

1. String, Number, Array 같은 표준 빌트인 객체도 [[Construct]] 내부 메서드를 갖는 생성자 함수이므로 extends 키워드를 사용하여 확장할 수 있다.

[예제 25-80]

```
// Array 생성자 함수를 상속받아 확장한 MyArray
class MyArray extends Array {
    // 중복된 배열 요소를 제거하고 반환한다: [1, 1, 2, 3] => [1, 2, 3]
    uniq() {
        return this.filter((v, i, self) => self.indexOf(v) === i);
    }

    // 모든 배열 요소의 평균을 구한다: [1, 2, 3] => 2
    average() {
        return this.reduce((per, cur) => pre + cur, 0) / this.length;
    }
}

const myArray = new MyArray(1, 1, 2, 3);
console.log(myArray);   // MyArray(4) [1, 1, 2, 3]

// MyArray.prototype.uniq 호출
console.log(myArray.uniq());    // MyArray(3) [1, 2, 3]
```

```
// MyArray.prototype.average 호출
console.log(myArray.average()); // 1.75
```

- Array 생성자 함수를 상속받아 확장한 MyArray 클래스가 생성한 인스턴스 Array.prototype과 MyArray.prototype의 모든 메서드를 사용할 수 있다.
- 이때 주의할 것은 Array.prototype의 메서드 중에서 map, filter와 같이 새로운 배열을 반환하는 메서드가 **MyArray 클래스의 인스턴스**를 반환한다는 것이다.

```
console.log(myArray.filter(v => v % 2) instanceof MyArray); // true
```

- 만약 새로운 배열을 반환하는 메서드가 MyArray 클래스의 인스턴스를 반환하지 않고 Array의 인스턴스를 반환하면 MyArray 클래스의 메서드와 **메서드 체이닝(method chaining)이 불가능**하다.

```
// 메서드 체이닝
// [1, 1, 2, 3] => [ 1, 1, 3 ] => [ 1, 3 ] => 2
console.log(myArray.filter(v => v % 2).uniq().average());   // 2
```

- myArray.filter가 반환하는 인스턴스는 MyArray 클래스가 생성한 인스턴스로 uniq 메서드를 연이어 호출(메서드 체이닝)할 수 있다.

  uniq 메서드가 반환하는 인스턴스는 Array.prototype.filter에 의해 생성되었기 때문에 Array 생성자 함수가 생성한 인스턴스로 생각할 수도 있겠다. 하지만 uniq 메서드가 반환하는 인스턴스도 MyArray 타입이다. 따라서 uniq 메서드가 반환하는 인스턴스로 average 메서드를 연이어 호출(메서드 체이닝)할 수 있다.

2. 만약 MyArray 클래스의 uniq 메서드가 MyArray 클래스가 생성한 인스턴스가 아닌 Array가 생성한 인스턴스를 반환하게 하려면 다음과 같이 **Symbol.species**를 사용하여 정적 접근자 프로퍼티를 추가한다.

```
// Array 생성자 함수를 상속받아 확장한 MyArray
class MyArray extends Array {
    // 모든 메서드가 Array 타입의 인스턴스를 반환하도록 한다.
    static get [Symbol.species]() { return Array; }

    // 중복된 배열 요소를 제거하고 반환한다: [1, 1, 2, 3] => [1, 2, 3]
    uniq() {
        return this.filter((v, i, self) => self.indexOf(v) === i);
    }

    // 모든 배열 요소의 평균을 구한다: [1, 2, 3] => 2
    average() {
        return this.reduce((per, cur) => pre + cur, 0) / this.length;
    }
}
```

```
const myArray = new MyArray(1, 1, 2, 3);

console.log(myArray.uniq() instanceof MyArray); // false
console.log(myArray.uniq() instanceof Array);   // true

// 메서드 체이닝
// uniq 메서드는 Array 인스턴스를 반환하므로 average 메서드를 호출할 수 없다.
console.log(myArray.uniq().average());
// TypeError: myArray.uniq( ... ).average is not a function
```